一种用于 Android 应用的反控制混淆系统 *

曹宏盛，焦　健，李登辉

(北京信息科技大学 计算机学院, 北京 100101)

摘　要：Android 恶意软件中的控制混淆技术，可以增加传统 Android 应用软件执行路径检测的难度，是目前代码静态分析的主要困难之一。针对该问题进行了研究，并设计系统 DOCFDroid 用于解决此问题。该系统在预处理阶段获取 CFG 关系矩阵，使用深度优先查找待分析路径集合；依据用户给定的源点集合和终节点集合，得到粗糙路径；然后采用权重筛选的算法，可以有效的获取目标路径集合。在实验阶段以 DroidBench 1.2 为基础构建测试样本集，验证该方法的有效性。实验结果表明，该方法能有效抵抗控制流混淆带来的干扰，目标路径识别率可达 95.31%。

关键词：反控制混淆；Android；控制混淆；路径查找

中图分类号：TP309.1　　doi: 10.3969/j.issn.1001-3695.2017.12.0759

Deobfuscation system for Android applications

Cao Hongsheng, Jiao Jian, Li Denghui

(*Computer School of Beijing Information Science & Technology University*, *Beijing* 100101, *China*)

Abstract: Control obfuscation can greatly increase the difficulty on detecting implementation path of Android application, and also is the one of the main difficulties on current code static analysis. This article had carried on the research to this question and designed a system DOCFDroid used to solve this problem. The system obtained the CFG relation matrix in the pretreatment stage, and used the depth-first algorithm to find the set of the path. According to the set of source points and the set of end points, the coarse path was obtained. The algorithm of weight screening could effectively obtain the target path set. Based on the sample set DroidBench1.2, the test sample set was further constructed to verify the validity of this method. The experimental results show that the system can effectively resist the interference caused by control obfuscation, and the recognition rate of target path can reach 95.31%.

Key Words: control flow deobfuscation; android; control flow obfuscation; path detection

0　引言

混淆技术的出现可以帮助隐藏代码内部的真实信息，最早应用于软件反分析和防盗版领域，但随着近年来恶意代码的泛滥，越来越多的黑客都会采用代码混淆技术来“躲避”检测工具。由于目前针对恶意代码的检测方法主要基于特征码的方式，混淆技术的大量使用可以有效隐藏自身的特征，躲避防病毒工具的查杀，加剧了分析过程的时空复杂性，造成特征码的数量日益庞大，使代码检测的时效性和准确都受到了严重的影响。

在通用的混淆技术中，目前比较常用且难以防范的是控制流混淆，其目的[1]是改变或复杂化程序的控制流，使程序更难以破译。控制混淆通过改变控制流图(CFG)的结构，达到混淆目的。按照其实方式控制混淆又可分为重打包、字节码控制和插入冗余分支等多种实现技术。文献[2,3]将控制混淆方法分为三类，分别是：计算混淆、排序混淆和聚合混淆，如表 1 所示。

表 1　控制混淆分类

计算混淆	内联法
	全局声明
	克隆法
	循环展开
排序混淆	声明重排序
	循环重排序
	表达式重排序
聚合混淆	循环条件扩展
	转变成不可还原流程图表解析

由于控制混淆的多样性和易于实现的特点。越来越多的恶意代码普遍采用控制混淆技术逃避安全分析和检测，针对恶意代码的反混淆技术已经成为代码安全的关注热点。

基金项目：网络文化与传播重点实验室—开放课题—移动互联网与网络安全研究（ICDDXN001）；中央引导地方科技发展专项资助项目（Z171100004717002）

作者简介：曹宏盛（1989-），男，广西钦州人，硕士研究生，主要研究方向为网络安全（caohsh@mail.bistu.edu.cn）；焦健（1978-），男，河北沧州人，副教授，博士，主要研究方向为网络测量、网络安全等；李登辉（1989-），男，河南漯河人，硕士，主要研究方向为网络安全.

Baumann 等人[4]提设计了一个 Anti-ProGuard 的一个原型实现，使用软件相似度算法检测混淆之后的恶意软件。文献[5]提出了一套抗混淆的恶意应用变种识别系统，包括行为分析、特征提取和恶意应用识别三个主要部分。其方法通过实现高透明度监控，分析代码行为，提取关键行为逻辑和抗干扰处理，实现对恶意应用变种的识别，但该文采用的动态污点分析，对资源和时间要求较大。文献[6]提出了检测系统 DroidChameleon，它能够对检测出重打包、字节码控制混淆技术的应用。文献[7]提出的检测工具 ViewDroid 同样也是针对 Android 平台重打包后的应用进行检测。文献[8]提出了一种抗混淆的 Android 应用相似性检测方法。其思路是提取 Android 应用内特定文件的内容特征计算应用相似性。文献[9]目的是检测恶意软件是否属于同一家族，使用 API 调用及数据依赖关系描述特征，结合语义和统计学习进行检测，降低了部分代码混淆技术干扰，但分析复杂度较高。文献[10]提出的检测工具 AppFence 结合了 MockDroid 和 TISSA，采用了与 TaintDroid 类似的针对污点分析方法，能够检测经过混淆、加密的信息泄露。这些文献对恶意应用的识别方面提出有效的方法，但并不能给出恶意软件混淆前后的功能实现路径，分析人员无法分析应用在混淆前后的行为和运行过程。文献[11]使用 CFG（Control Flow Graph:控制流图）表示行为为特征，通过分析同构 CFG 对抗部分混淆，该文引用 CFG 来解决混淆问题，但还是没有给出恶意应用混淆前后的功能实现路径。

从目前的研究结果可知，基于特征识别的传统方法已经无法满足现有反混淆需求，需要以行为特征为主要的分析方法，通过识别恶意应用的实际行为，达到抵抗混淆技术干扰的目的。文献[2]给出混淆的定义特别指出代码混淆不改变原程序的功能，因此混淆前后总能有一条实现程序功能的路径。本文以此为基础开展相应的研究，引用了 source-sink 机制，利用敏感函数定位 source 和 sink，并构建了恶意应用的 CFG，最后检测出目标路径（实现原功能）。

1 问题分析

控制流图由基本块和有向边两类元素组成。对于程序原有的某些特定功能，总存在能从起始节点到达终止节点的路径。因此，对混淆之后的应用，查找实现原有功能的执行路径是可行的。为了便于说明，构建了如图 1 所示的 CFG 控制混淆处理前后的示例，图中的圆形表示 CFG 中的基本块，有向边表示控制流，其中标志为 S 的基本块为起始基本块；标志为 E 的基本块为终结基本块。

图 1 的 normal 所示，可达路径有 path1={S,1,2,3,E}，path2={S,1,4,2,3,E}，path3={S,1,4,5,E}和 path3={S,1,4,5,E}，则对应的路径集合为 paths={path1, path2, path3}，可知 paths 中皆为可达路径。假设该应用输入数据为 in，执行结果为 out，数据作用于 paths 的可达路径必能够得到执行结果 out。

如图 1 的 obfuscate 所示为经过控制混淆后所得的 CFG，产生了不可达路径 path4={S,1,6,7}、path5={S,1,2,7}、path6={S,1,2,7,8} 和 path7={S,1,2,8,9}，而 path1 变为 path8={S,1,2,8,3,E}，path2 变为 path9={S,1,4,2,8,3,E}，最终的路径集合为 paths'={ path3, path4, path5, path6, path7, path8, path9}。不可达路径集合 $path_{invalid}$={path4, path5, path6, path7}给逆向分析工作制造障碍。

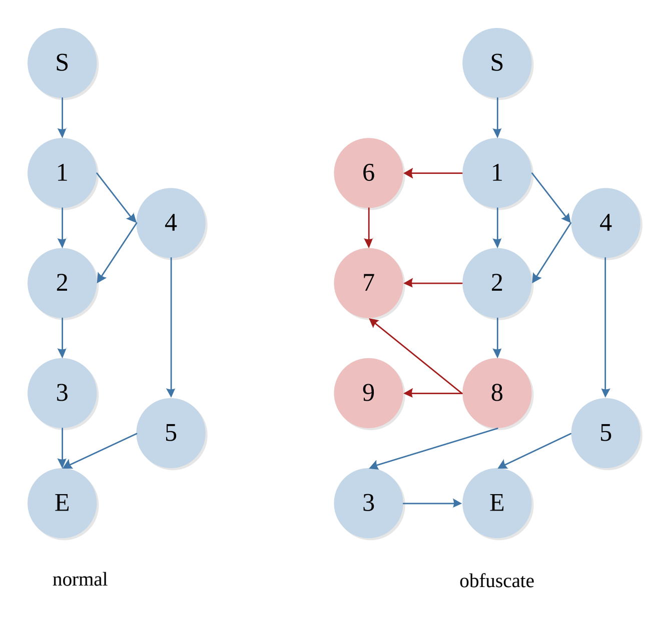


图 1　示例控制流图

其中 paths 和 paths'是同一个应用对应的两个不同的路径集合，paths'是根据混淆后代码的 CFG 求得的路径集合。根据 source-sink 机制，其中 n_s 表示该基本块包含源点，下标 s 属于源点集合 Source，n_d 表示该基本块包含终节点，下标 d 属于终节点集合 Sink，因此，反混淆的本质问题是如何从 paths'中尽可能地还原出 paths。即表示从代码中找到从 n_s 到 n_d 的可达路径 pathsd 构成的集合（记作 $Path_{source_sink}$）。

$path_{sd}$ 根据函数分为两类：一类 n_s 和 n_d 在同一个函数中；另外一类是 n_s 和 n_d 分别在不同函数中，具体可以分为两种情况：

a）n_s 和 n_d 中的变量，在本函数的其他基本块中也有操作处理，通过 n_s 和 n_d 中的变量，分别查找 n_s 和 n_d 所在函数中关于其变量的路径；

b）n_s 或 n_d 中的变量，在本函数的其他基本块中没有操作处理，输出 n_s 和 n_d 所在的函数名以及基本块序号。

2 系统设计

2.1 预处理阶段

基于 CFG 的 Android 应用反控制混淆方法的系统（如图 2 所示，在系统地第一阶段主要完成数据预处理 ，在 1.1 中，系统将指定的安装文件，通过反编译抽取出其中 class.des 和 mainfirst.xml 文件，并根据这些文件，生成 SSA (Static Single-Assignment:静态单赋值) 文件。1.2 阶段使用 SSA，并根据用户的需求，生成相应的 CFG。

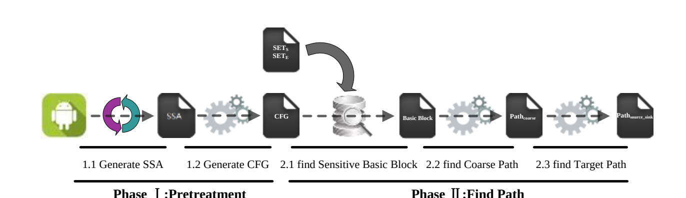


图 2　系统结构图

CFG 可以将其表示成矩阵形的形式，称为 CFRM（Control Flow Relation Matrix：控制流关系矩阵），二者形式定义如下：

$CFG=(E,N)$，其中 E 为有向边集合，N={M，V，t}为基本块（图中的节点）集合，其中 M 为方法/函数集合，V 为变量集合，t 为分支数，其中 t≥0。

$CFRM=(R,N)$，N 是 CFG 中基本块集合；R 是一个二维矩阵，即从 NxN 到{1，0}的映射，称为 N 上的二元关系；$\forall(n_i,n_j)\in N\times N$。

2.2　查找路径阶段

系统的第二阶段主要作用是根据预处理模块得到的 CFG，检测出符合用户当下需求的路径(功能路径)。具体分为三个部分：

2.2.1 查找敏感基本块

敏感基本块查找功能（如图 2 的 2.1 所示）。在查找敏感基本块之前，先在预处理得到的 CFG 的基础上，生成 CFRM。

设计算法 *CONSTRAINT_VALUE*。该算法找出所有包含源点元素以及终节点元素的敏感基本块集合(起始基本块集合和终结基本块集合)，并取两者中的元素个数大的值作为路径条数约束值 $r_{constraint}$。

算法的输入分别是：CFRM，SET_s 源点元素集合以及 SET_d 的终点元素集合，两者都为一些特征集，用于匹配查找敏感基本块；输出：约束值 $r_{constraint}$。

以下是该算法的伪代码：

CONSTRAINT_VALUE(CFRM, SET_s, SET_d)

$s=0$

$d=0$

$N_d=\varnothing$

$N_s=\varnothing$

for each n ∈ CFRM.N　//遍历 CFG 中的基本块

　for each m ∈ n.M　//遍历基本块中的表达式

　　if m ∈ SET_d　//表达式含有终点元素

　　　$N_d=N_d\cup n$

　　　$s+=1$

　　else if m ∈ SET_s　//表达式含有源点元素

　　　$N_s=N_s\cup n$

　　　$d+=1$

store N_s

store N_d

return max(s, d)

算法通过遍历 *CFRM*，根据 SET_s 源点元素集合以及 SET_d 的终点元素集合，将找到的敏感基本块分别保存，这些敏感基本块用于在后面粗糙路径查找中定位 source-sink，通过定位 source-sink，缩小了图查找的范围，有益于提高查找的效率。最后 *max(s, d)*函数来返回约束值，用于限定最后查找到的目标路径数量。

2.2.2 查找粗糙路径

系统的 2.2 阶段主要实现查找粗糙路径。以图 1 中的 CFG 为例，混淆后的路径集合：

paths'={ path3, path4, path5, path6, path7, path8, path9}。

根据 n_s 和 n_d 的元素组合，使用深度优先算法对 *paths'*集合进行裁剪，得到粗糙路径集合 $Path_{coarse}$，记作：

$$Path_{coarse}\subseteq Path',\forall p_i\in Path_{coarse},$$
$$\exists n_j\in N_s,n_k\in N_d,j\le k,$$

算法的输入是：CFRM，起始基本块集合 N_s 以及终结基本块集合 N_d；输出为：$Path_{coarse}$。

算法伪代码如下：

GENERATE_COARSE_PATH_SET(CFRM, N_d, N_s)

$PATH_{tmp}=\varnothing$

$Path_{coarse}=\varnothing$

for each n ∈ N_d

//使用深度优先查找包含终结基本块的路径

$PATH_{tmp}$= $PATH_{tmp}$∪GENERATE_DFS(CFRM.R, n)

for each p ∈ $PATH_{tmp}$

//在 $PATH_{tmp}$ 基础上查找包含起始基本块的路径

　for each m ∈ p

　　if m ∈ N_s

　　　$Path_{coarse}=Path_{coarse}\cup p$

　　　break

return $Path_{coarse}$

通过上一小节得到的敏感基本块，使用深度优先算法遍历 CFRM.*R* 二维矩阵得连通 source 和 sink 的路径，并存到 $Path_{coarse}$。$Path_{coarse}$ 是整个 CFG 路径的子集，该算法进一步缩减了裁剪范围，获得更精确的目标路径。

以图 1 中的 CFG 为例，　混淆后的路径集合：

paths'={ path3, path4, path5, path6, path7, path8, path9}。

经过算法处理 *paths'*路径集合的 CFG，得到粗糙路径集合 $Path_{coarse}$ 的 CFG，如图 3 所示。

从图 3 可以看出，包含基本块 6、7、9 的路径，并没有同时包含起始基本块(S 基本块)和终结基本块(E 基本块)。通过对粗糙路径查找，将这些无关基本块剔除，得到粗糙路径集合，表现如图 3 的 $Path_{coarse}$ 的 CFG 所示。

2.2.3 查找目标路径

系统 2.3 阶段主要功能，通过算法 *OPTIMIZE_PATH_SET*()，计算各个粗糙路径的权重，然后根据权重筛选粗糙路径 $Path_{coarse}$ 得到目标路径 $Path_{source_sink}$。

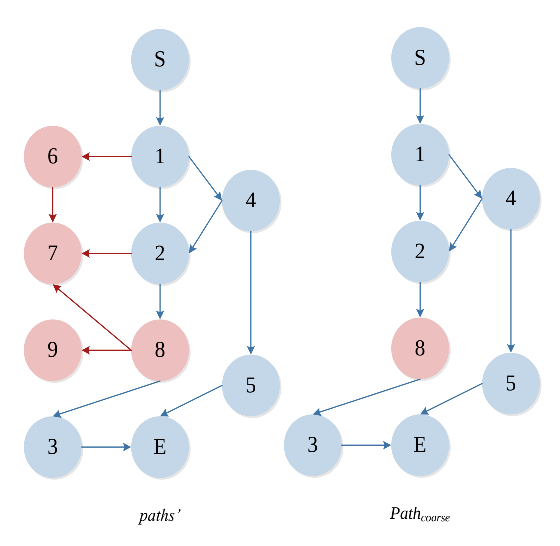


图 3　示例混淆路径集合 *paths'*剪枝

该算法主要分成三个步骤：

a)从粗糙路径集合中查找出每条路径的起始基本块以及终结基本块中的所有变量，伪代码中表示为 *FIND_RELATED_VAR(p)*，然后为变量赋予权值 1；

b)计算每条粗糙路径 p 的每一个基本块的权重，将这些权重相加得到路径 p 的权重，并将路径 p 的权重添加到权值集合 W 中；

c)将 W 集合中的元素按照从大到小排序，按照约束值 $r_{constraint}$ 选取出前 t 条权值大的路径作为目标路径。

根据算法 2.1，由 N_S（起始基本块集合）和 N_d（终结基本块集合）元素的个数，得到的约束值 $r_{constraint}$ 取值为 1，得到目标路径集合 $Path_{source_sink}$。如图 4 所示，对 $Path_{coarse}$ 做路径权重排序之后，再结合 $r_{constraint}$=1，优先选择第一条权重大的路径作为目标路径，得到目标路径集合 $Path_{source_sink}$。

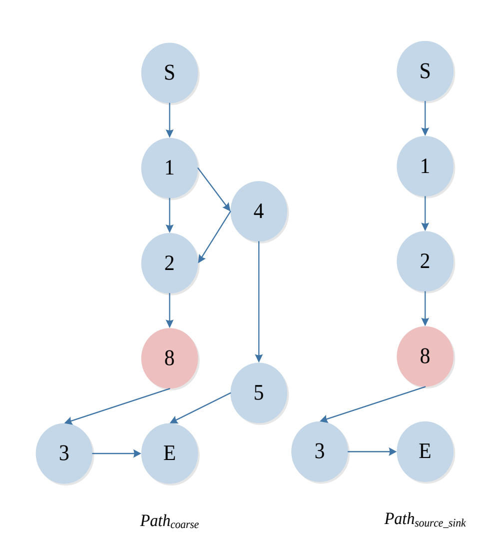


图 4　示例粗糙路径集合 $Path_{coarse}$ 剪枝

算法的输入分别是粗糙路径集合 $Path_{coarse}$，以及约束值 $r_{constraint}$；算法的输入出为目标路径 $Path_{source_sink}$。　算法伪代码如下所示：

OPTIMIZE_PATH_SET($Path_{coarse}$, $r_{constraint}$)

$W = \varnothing$

$PATH_{source_sink} = \varnothing$

for each p ∈ $Path_{coarse}$

　$w_p = 0$

//获取路径中的敏感变量集合

　v = FIND_RELATED_VAR(p)

　for each n ∈ p

//敏感变量在路径中出现的次数，计算路径权值

　　w_p+ = Type(n)*IsVarExist(v, n)*w_v

　$W = W \cup \langle w_p, p \rangle$

//权值排序

$W' = \text{DESCEND_SORT}(W)$

//由约束值获取前 t 个路径，作为目标路径

$Path_{source_sink} = Path_{ource_sink} \cup \{p_i \mid \langle w_p, p_i \rangle \in W', i \le r_{constraint}\}$

return $Path_{source_sink}$

该算法从表达式和变量这两个层次去给得到的粗糙路径进行权重计算和优先级划分。通过敏感变量计算路径权重的方式，这一思路能够有效并快速地划分各个粗糙路径的优先级，从而得到目标路径。

3　实验验证

为验证算法在检测控制混淆后的 Android 应用路径的可行性与有效性，设计并实现了系统 DOCFDroid，开发工具基于 Eclipse4.5.2、JDK1.8、基础框架 Soot。检测系统中识别 source 和 sink 所采用的特征集来源于文献[12]。

3.1　DOCFDroid 功能展示

本文实验采用的样本集为：混淆后的 DroidBench 1.2 样本集[13]：在 DroidBench 1.2 数据集的基础上，使用混淆技术进一步构建的样本集。DroidBench 1.2 的 59 个开源的 Android 应用共包含 52 个人工注入的信息泄露漏洞，包括数组与列表的下标混淆，ICC（Inter-component Communication：进程通信组件）、代码混淆、反射以及隐式泄露等，已知漏洞所分布的数量和位置，并且包含一些误报陷阱，用于评估检测系统的误报率。该数据集已成为 Android 信息泄露检测公认数据集，且围绕该数据集已进行了多项高水平研究。本文在 DroidBench 1.2 数据集的基础上，进一步使用混淆技术构建本实验的测试样本数据集。

实验方法如下：使用多种控制混淆方法包括，循环条件扩展，插入冗余分支等，人为混淆样本集代码。样本集 CFG 混淆前后及路径查找对比如图 5、图 6 和图 7 所示。

如图 5 所示。左边的图是由样本"PrivateDataLeak1.apk"中泄漏信息的函数源码生成的 CFG，该 CFG 展示了该函数的所有基本块及其代码中间表示；图右边为未混淆的软件源代码，可以看到代码整体是一个 if 语句，并且 if 语句中只有一个 for 循环语句分支。该代码主要通过获取用户的日志数据和手机消

息，并把这些数据发送至远程服务器，达到目标信息泄漏的目的。

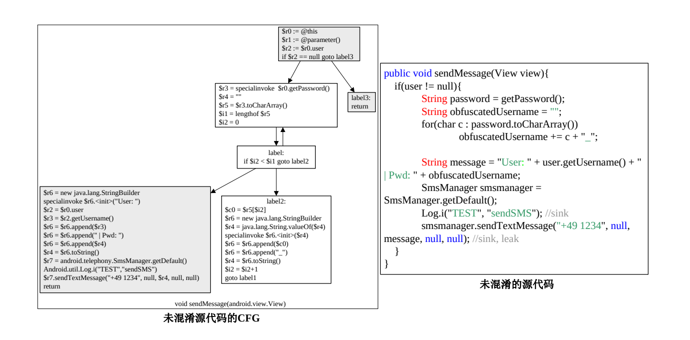


图 5　未混淆的源代码及其 CFG

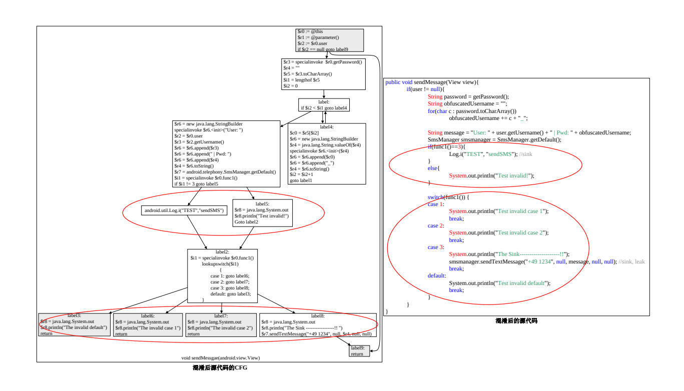


图 6　混淆后的源代码及其 CFG

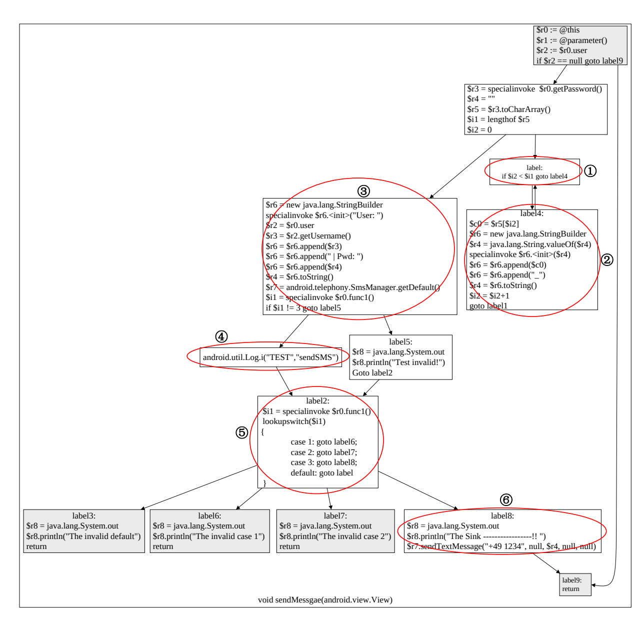


图 7　路径查找

如图 6 所示。图右边为混淆后的软件源代码，在源代码，本文插入了一个冗余的 switch 语句分支和 if 语句分支，使函数原来实现功能的路径变得复杂，如椭圆标志部分所示。两个语句中用于做条件判定的函数 func1()永远取值为 3，因此程序还是执行原来的代码，函数功能未改变。但是，达到了混淆代码的目的，使得代码分析增加了难度。系统 DOCFDroid 通过检测混淆后的应用，得到该应用源代码的 CFG（图 5 左边），与图 5 比对，可以看到图中多出一些分支，如图中椭圆标志部分所示。

系统 DOCFDroid 处理之后到的目标路径(图 7 椭圆圈出的基本块组成)，一样能够实现“PrivateDataLeak1.apk”通过获取用户的日志数据和手机消息，并把这些数据发送至远程服务器，达到目标信息泄露的目的。

经反控制混淆路径查找之后，找到了可以实现函数原本功能的基本块。这些基本块可能并不与原来的基本块数量完全一致，但是这些基本块组合而成的路径同样能够实现函数原本的功能。如图中椭圆标志部分所示，基本块①、②对应未混淆源代码的 for 循环语部分；基本块③为变量“message”的赋值；基本块④对应代码“Log.i("TEST", "sendSMS");”，用于获取日志数据；基本块⑤为本文插入的 switch 语句；基本块⑥对应未混淆的源代码“smsmanager.sendTextMessage("+49 1234", null, message, null, null); ”，用于实现信息数据的发送，达到目标数据泄露的目的。

3.2　性能比较

DroidBench 1.2 的实验结果如表 2 所示。从结果可以看出，未经过混淆的 59 个样本数据集总共有 64 条目标路径。实验对此 59 个混淆之后的 APK 文件进行路径检测实验，可识别出 61 条目标路径，识别率为 95.31%，其中 Library1（只有 source）和 LogNoLeak（只有 sink）不符合本文的目标路径条件，看成无效样本数据。如表 2 所示，漏报率为 4.69%，路径漏报的样本数据集有 ImplicitFlow2、ImplicitFlow3、ImplicitFlow4。 分析原因为在计算路径权值时，未能找到对应变量，重复对首个路径变量赋值，导致路径筛选失败。实验的误报率为 6.25%，路径误报的样本数据集有 LocationLeak3、Button2 和 ImplicitFlow1。分析发现，使用的污点源识别库导致部分识别出来的起始基本块和终结基本块，多于样本集中标注的 source 和 sink，导致目标路径个数比原样本集数据中的路径更多。

4　结束语

目前的研究大多基于特征识别的传统方法，该方法已经无法满足现有反混淆需求，通过行为特征为主要的分析方法，识别恶意应用的实际行为，达到抵抗混淆技术干扰的目的。本文提出并实现了反控制混淆系统 DOCFDroid，检测出混淆前后应用功能的实现路径。本文的主要研究工作集中在两点：第一、依据混淆的定义[2]以及控制混淆[1]目的，论述对控制混淆后的软件路径检测的可行性，将反控制混淆问题转变为控制流图(control flow graph)的路径查找问题，从图论的角度提出了反控制混淆的方法;第二，利用源点和终节点集合确定目标路径的起始节点和终止节点，简化并有效地提高粗糙路径的检测；使用权重筛选的算法进行路径优化，去除冗余粗糙路径，提高目标路径的识别率，实验表明，系统对目标路径的识别率达到 95.31%。对检测控制混淆后的 Android 数据集时，能够有效识别出目标路径，有效降低误报与漏报。

下一的研究内容主要包括：a）完善路径权重计算算法，进

一步降低由路径变量查找失败导致的漏报率；b）筛选或者构建更完善的 source、sink 识别库，进一步降低的误报率。

表 2 DroidBench 1.2 数据集实验结果

样本名	原始目标路径条数	混淆后目标路径条数	处理后路径条数	准确识别路径条数	样本名	原始目标路径条数	混淆后目标路径条数	处理后路径条数	准确识别路径条数
DirectLeak*	1	48	1	1	ObjectSensitivity2	1	231	1	1
InactiveActivity	1	1331	1	1	Exceptions1	1	21	1	1
Library1	0	231	0	0	Exceptions2	1	21	1	1
Library2	1	231	1	1	Exceptions3	1	21	1	1
LogNoLeak	0	231	0	0	Exceptions4	1	21	1	1
Obfuscation1	1	231	1	1	Loop1	1	252	1	1
PrivateDataLeak1	1	2783	1	1	Loop2	1	234	1	1
PrivateDataLeak2	1	231	1	1	SourceCodeSpecific1	1	231	1	1
PrivateDataLeak3	2	231	2	2	StaticInitialization1	1	231	1	1
ArrayAccess1	1	231	1	1	UnreachableCode	1	231	1	1
ArrayAccess2	1	231	1	1	ImplicitFlow1	1	231	2	1
HashMapAccess1	1	231	1	1	ImplicitFlow2	2	231	1	1
ListAccess1	1	231	1	1	ImplicitFlow3	2	462	1	1
AnonymousClass1	1	231	1	1	ImplicitFlow4	2	2772	1	1
Button1	1	231	1	1	IntentSink1	1	231	1	1
Button2	1	231	3	1	IntentSink2	1	231	1	1
Button3	1	231	1	1	ActivityLife1	2	231	2	2
LocationLeak1	1	231	1	1	ActivityLife2	1	231	1	1
LocationLeak2	1	231	1	1	ActivityLife3	1	231	1	1
LocationLeak3	1	231	2	1	ActivityLife4	1	231	1	1
MethodOverride1	1	231	1	1	ApplicationLife1	1	231	1	1
MultiHandlers1	2	231	2	2	ApplicationLife2	1	231	1	1
Ordering1	1	231	1	1	ApplicationLife3	2	231	2	2
Unregister1	1	231	1	1	BroadcastLifecycle1	1	231	1	1
FieldSensitivity1	1	231	1	1	SeviceLifecycle1	1	21	1	1
FieldSensitivity2	1	231	1	1	Reflection1	1	231	1	1
FieldSensitivity3	1	231	1	1	Reflection2	1	231	1	1
FieldSensitivity4	1	231	1	1	Reflection3	1	231	1	1
InheritedObjects1	1	231	1	1	Reflection4	1	231	1	1
ObjectSensitivity1	1	231	1	1					
总计	**30**	**10399**	**33**	**30**		**34**	**8445**	**32**	**31**
原始目标路径总数：64				**处理后路径总数：65**			**准确识别路径总数：61**		
识别率					**95.31%**				
误报率					**6.25%**				
漏报率					**4.69%**				

参考文献：

[1] 王建民，余志伟，王朝坤，等. Java 程序混淆技术综述 [J]. 计算机学报, 2011, 34 (9): 1578-88.

[2] Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations [R]. Auckland: University of Auckland, 1997.

[3] Hoffmann J, Rytilahti T, Maiorca D, et al. Evaluating analysis tools for android apps: status quo and robustness against obfuscation [C]// Proc of the

6th ACM Conference on Data and Application Security and Privacy. New York: ACM Press, 2016.

[4] Baumann R, Protsenko M, Müller T. Anti-ProGuard: towards automated Deobfuscation of Android apps [C]// Proc of the 4th Workshop on Security in Highly Connected IT Systems. New York: ACM Press, 2017.

[5] 王蕊，苏璞睿，杨轶，等. 一种抗混淆的恶意代码变种识别系统 [J]. 电子学报, 39 (10): 2322-2330, 2011.

[6] Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks [C]// Proc of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, New York: ACM Press, 2013: 329-334.

[7] Zhang F, Huang H, Zhu S, et al. ViewDroid: towards obfuscation-resilient mobile application repackaging detection [C]// Proc of ACM Conference on Security and Privacy in Wireless & Mobile Networks. New York: ACM Press, 2014: 25-36.

[8] 焦四辈，应凌云，杨轶，等. 一种抗混淆的大规模 Android 应用相似性检测方法 [J]. 计算机研究与发展, 2015, 51 (7): 1446-1457.

[9] Christodorescu M, Jha S, Kruegel C. Mining specifications of malicious behavior [C]// Proc of the 1st India Software Engineering Conference. New York: ACM Press, 2008: 5-14.

[10] Hornyack P, Han S, Jung J, et al. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications [C]// Proc of the 18th ACM Conference on Computer and Communications Security. New York: ACM Press, 2011: 639-652.

[11] Bonfante G, Kaczmarek M, Marion J Y. Architecture of a morphological malware detector [J]. Journal in Computer Virology, 2009, 5 (3): 263-270.

[12] Rasthofer S, Steven A, Bodden E. A machine-learning approach for classifying and categorizing android sources and sinks [C]// Proc of Network and Distributed System Security Symposium. 2014.

[13] Fritz C, Arzt S, Rasthofer S. DroidBench [EB/OL]. (2015) [2017]. https://github. com/secure-software-engineering/DroidBench.